CSE 473 Project 2: Memory Management

(due Tue, November 8 by 11:59PM thru Angel - NO EXTENSIONS WILL BE GIVEN!)

Please direct all your project-related questions/clarifications to the TAs, either in person or by email. Piazza is also available for queries (but no exchange of code is permitted).

UPDATED ON: 27-OCT-2016.

Description

In this project, you will implement four memory allocation schemes to manage a chunk of memory and mimic `malloc()` and `free()` functions for a single thread system:

1. First Fit
2. Best Fit
3. Worst Fit and
4. Buddy System.

The `main()` in the testcases will first use a standard glibc `malloc()` call to allocate a large amount of memory from the system. The specific amount of memory that your system will handle will be defined by a parameter `mem_size` which is a parameter of the `setup()` function. Your code will then use this memory to form and return the requested smaller chunks of memory to the subsequent `my_malloc()` calls.

For the specifics and any latest updates make sure you keep checking this web-site for any latest information/updates regarding the project.

Deliverables

Your project should implement the following interfaces in C programming language, and all these four schemes should be provided in a new file called my_memory.c.

```
void setup(int malloc_type, int mem_size, void* start_of_memory);
```

This function will be invoked once in the beginning of execution and is to be used for two purposes:

1. Initializating your implementation to the type of memory management scheme using the parameter `malloc_type`. It will contain values from 0 to 3, denoting 0 for First-Fit, 1 for Best-Fit, 2 for Worst-Fit and 3 for Buddy System.
2. Providing with a contigous chunk of memory, whose starting address is passed in the argument, `start_of_memory` pointer and the size of this chunk is supplied by the parameter `mem_size`.

The memory management schemes should use this chunk of memory pointed by `start_of_memory` for all subsequent my_malloc() allocations.

```
void* my_malloc(int size);
```

This function when invoked, should allocate a chunk of memory and return a pointer to the starting of the allocated memory chunk (which should be in the range from `start_of_memory` to

`start_of_memory + mem_size`. The size of allocation is controlled by the argument **`size`**. Your implementation of this function should perform the necessary logic to track and maintain the allocated memory chunks, the free holes and their respective sizes so that it does not allocate the same address to more than one **`my_malloc()`** callees (before the address gets freed). Also, when the requested size is not allocatable this function should return `(void *)-1`.

`void my_free(void* ptr);`

This function deallocates and frees a pointer allocated using a previous **`my_malloc()`** call and hence creates a hole. The logic here should also take care of merging adjacent holes into bigger holes. To do so, you need to track the size allocated for each pointer returned from **`my_malloc()`**, in a 4-byte header before the pointer returned. A detailed explanation is given below.

`int num_holes();`

This function should return the number of holes that are in the system at the time of invoking the function. Follow these points for counting number of holes.

1. A hole of size 0 is no longer a hole!
2. In buddy system, you should NOT count/fuse two non-buddies that are still adjacent to each other into a single bigger hole. Only buddies can be fused to form bigger holes.

`int num_free_bytes();`

This function should return the total number of free bytes (i.e. sum of all hole sizes) in the system at the time of invoking this function.

Storing the allocated size as header:

The memory management subsystem services memory requests by matching the size in the **`my_malloc()`** request with a large enough hole to satisfy the request.
The **`size`** parameter is passed as argument only in **`my_malloc()`** and NOT in **`my_free()`**. However it is necessary to know the size of what is being freed to keep track of "free holes" and implement compaction mechanisms.
The way it is usually handled is that each allocated memory block has a header inside it. It stores the size of the block (as an integer) which should be 4 bytes. When a block is to be allocated in the **`my_malloc()`**, a pointer to the byte after the header of that block is returned. The header information should be used when the user calls **`my_free()`** to find out the size of what is being freed, and return the block to the freed pool. Free chunks/holes should be fused together with its neighbors as dictated by the respective allocation scheme.

Resources Provided

The links below direct you to various files/man pages that are vital for this implementation.

Boiler-plate source files

You are provided with a skeleton file here to help you get started with the implementation. To test the expected behavior of your source code follow the steps described below.

Debugging your code with test cases

For testing/debugging your work, a set of input files and their corresponding output files are provided **here**. There are 12 source files numbered as test<number>.c and a header file named memalloc.h.

NOTE: You are NOT supposed to modify test*.c and memalloc.h, or modify any of the input/output formatting mechanisms. The inputs and sample outputs are given just for illustrative purposes to test your code. You can create more extensive input files for further testing. The TAs may surprise you with several other test inputs during the demo and your routines should still work.

Compiling and Running

To compile your code (my_memory.c, test<number>.c and memalloc.h) we provide a sample Makefile here. Download and run the Makefile script using the command 'make input=<number>'. This will result in the generation of the binary 'test-out', which will be run and an output file named test<number>_output.txt will get created. This output file will then get compared with the expected output in TestOutputs/test<number>_output.txt file and a diff of the two files will be printed on the screen.
As you can see in the Makefile, the default input <number> is 1.
To run the binary yourself, use the command: `./test-out`

Please stay tuned constantly to the angel page for the exact and latest interface functions you need to implement, their arguments, test programs, examples, documentation/manuals and announcements.

The projects (including a **brief** report) is due by 11:59PM on the designated day and the reports+programs should be turned in using the Submission Guidelines. NO EXTENSIONS WILL BE ENTERTAINED.

You need to set up an appointment with your TAs to demonstrate your implementation. You can work in teams (at most 2 per team - they could be across sections), or individually, for the project. You are free to choose your partner (and can be different from that of Project 1) but if either of you choose to drop out of your team for any reason at any time, each of you will be individually responsible for implementing and demonstrating the entire project and writing the project report by the designated deadline.
If you have difficulty finding a partner, give your name to the TAs who will maintain an online list of students without partners to pair you up. Even though you will work in pairs, each of you should be fully familiar with the entire code and be prepared to answer a range of questions related to the entire project. It will be a good idea to work together at least during the initial design of the code. You should also ensure that there is a fair division of labor between the members.

No form of academic dishonesty of any kind will be tolerated. All projects should be individually undertaken by each team to the best of that team's ability, without seeking any external help. No exchange of code is permitted between teams. Severe penalties will be imposed not just in the project grade, but the dishonestry will be reported to the University\92s Office of Student Conduct for possible further academic sanctions.

Test files

Full tarball

Submission Guidelines

Piazza